Programmation (fonctionnelle)
Organisation et compilation de projets

Avant-propos

Jusqu'à présent, en Java et OCaml, on vous a mentionné l'existence de librairies extérieures au langage. On a juste négligemment oublié de vous expliquer comment utiliser ces librairies, parce que c'est quelque chose d'assez horrible. Maleureusement, vous n'avez pas le choix. Du coup, voici quelques conseils – et quelques exercices – sur la gestion de librairies extérieures et l'organisation de projets.

Notez que l'essentiel de cette fiche s'applique à tous les langages de programmation, pas juste à OCaml. Notez aussi qu'il y a d'autres méthodes plus simples, que nous verrons peut-être plus tard ce semestre.

Rappels sur la compilation en OCaml

Compilation de modules

Pour définir un module `Mon_module`, vous avez besoin de

une interface (optionnelle).
un fichier `mon_module.mli`, qui contient la liste des types et des valeurs définies par le module

une implantation.
un fichier `mon_module.ml`, qui contient les types et les valeurs définies par le module.

Compilation de l'interface

Pour compiler l'interface, on commencera par taper, dans l'invite de commande,

```
ocamlc -c mon_module.mli
```

Ceci invoque le compilateur `ocamlc`, en lui demandant à l'aide de `-c`, de procéder à une compilation séparée. La commande produira le fichier `mon_module.cmi`, l'*interface compilée*. Ce fichier est suffisant pour commencer à travailler sur d'autres modules qui pourraient utiliser `Mon_module`. Si le fichier `mon_module.mli` n'existe pas, oubliez cela. OCaml se passera du fichier `.cmi`.

Compilation de l'implantation

Pour compiler l'implantation, commencez par compiler l'interface, si elle existe.

Ensuite, dans l'invite de commande,

```
ocamlc -c mon_module.ml
```

Ceci invoque le même compilateur, et produit le fichier `mon_module.cmo`, le module compilé lui-même. Si votre module `Mon_module` utilise d'autres modules `Module_a`, `Module_b`, `Module_c`, les fichiers `modules_a.cmi`, `module_b.cmi` et `module_c.cmi` doivent déjà avoir été compilés (et être présents dans le répertoire).

Plus court

On peut compiler tout cela d'un coup, avec

```
ocamlc -c mon_module.mli mon_module.ml
```

Compilation de bibliothèques

Une fois que tous les fichiers `.cmo` sont compilés, on peut les joindre en une bibliothèque `ma_librairie.cma`. Ceci n'est jamais obligatoire.

Pour ce faire,

```
ocamlc -a mon_premier_module.cmo mon_deuxieme_module.cmo -o ma_librairie
```

Liaison d'un programme

Un programme est un ensemble de modules et de bibliothèques externes. Une fois que tous les fichiers `.cmo` sont compilés, on peut les joindre en un exécutable `mon_programme.exe` à l'aide de la commande

```
ocamlc une_librairie.cma mon_module_a.cmo mon_module_b.cmo -o
mon_programme.exe
```

Vous devez mettre dans cette liste *tous* les modules et *toutes* les bibliothèques utilisées, *par ordre de dépendances*, c'est-à-dire en mettant toujours le module/la bibliothèque utilisé *avant* le module qui l'utilise. Cette liste détermine dans quel ordre les valeurs sont calculées – c'est-à-dire dans quel ordre les modules sont exécutés. Vous en déduirez qu'il n'est pas possible d'écrire deux modules qui s'utilisent mutuellement.

Fichier de commandes Windows

Si vous avez une liste de commandes de l'invite de commande, vous pouvez en faire un mini-programme, qu'on appelle un *fichier de commandes* ou *script shell*. Pour ce faire, inscrivez toutes ces commandes, une par ligne, dans un fichier `mon_fichier.cmd`. Vous pourrez lancer le programme en tapant

`mon_fichier.cmd`
dans l'invite de commande.

Pour plus d'informations sur les fichiers de commandes Windows, n'hésitez pas à chercher de la documentation sur Internet.

Projet simple

Vous trouverez dans le répertoire `Echange L2` le fichier `demos_liste_triee.zip`, qui contient un exemple de manipulation de listes triées.

Exercice 1. Débrouillez-vous pour faire compiler et exécuter l'ensemble de ce projet. Notez la liste des commandes nécessaires pour cette compilation, vous en aurez besoin pour l'exercice 7.

Projet complexe

Lorsque le projet devient complexe, il devient nécessaire d'organiser les fichiers. Une méthode fréquemment utilisée (par exemple par les programmeurs de Linux, de OCaml, de Firefox ou encore d'OpenOffice) est de classer de la manière suivante :

- répertoire `nomduprojet`, contenant tous vos fichiers
  - fichier `INSTALL`, contenant les instructions nécessaires pour installer le programme
  - fichier `COPYING`, contenant les détails de la license d'utilisation du programme
  - fichier `README`, contenant les informations sur la version actuelle
  - fichier `ChangeLog`, contenant la liste de toutes les modifications depuis la création des fichiers
  - fichier `make.cmd`, un fichier de commandes qui va compiler tous vos fichiers sources dans le répertoire `src` et produire un exécutable dans le répertoire `nomduprojet` – ce qui permet notamment de ne pas avoir à tout recompiler à la main à chaque fois
  - fichier `make_clean.cmd`, un fichier de commandes qui va effacer tous les fichiers temporaires et l'exécutable – ce qui permet notamment de distribuer un projet nettoyé de tous ces fichiers
  - répertoire `src`, contenant tous vos fichiers source (en OCaml, les `.ml` et et `.mli`)

- répertoire `doc`, contenant toute la documentation
  - répertoire `manual`, contenant le manuel d'utilisation (si possible au format `pdf` ou `html`)
  - répertoire `api`, contenant la documentation générée automatiquement (en OCaml, par `ocamldoc` ou `ocamlweb`)
- répertoire `data`, contenant les images et autres ressources
- répertoire `lib`, contenant toutes les librairies externes nécessaires pour faire fonctionner votre projet (en OCaml, il s'agira généralement de fichers `.cma` et `.dll`).

Les exemples qui suivent emploient ce genre de structure. Vous êtes encouragés à faire de même pour votre projet.

Exercice 2. Vous trouverez dans le répertoire `Echange L2` le fichier `reines_texte.zip`, qui contient un exemple de résolution du problème des n reines. Débrouillez-vous pour faire compiler et exécuter l'ensemble de ce projet. Notez la liste des commandes nécessaires pour cette compilation, vous en aurez besoin pour l'exercice 7.

Les trois exercices qui suivent peuvent être faits dans n'importe quel ordre :

Exercice 3. Vous trouverez dans le répertoire `Echange L2` le fichier `reines_tk.zip`, qui contient un exemple de résolution du problème des n reines. Cette version utilise la bibliothèque LablTK. Vous aurez donc besoin de consulter la documentation de LablTK (et probablement d'installer quelque chose) avant de pouvoir avancer. Débrouillez-vous pour faire compiler et exécuter l'ensemble de ce projet. Notez la liste des commandes nécessaires pour cette compilation, vous en aurez besoin pour l'exercice 7.

Exercice 4. Vous trouverez dans le répertoire `Echange L2` le fichier `reines_gl.zip`, qui contient un exemple de résolution du problème des n reines. Cette version utilise la bibliothèque LablGL, que vous aurez besoin d'installer, et dont vous aurez besoin de consulter la documentation. Débrouillez-vous pour faire compiler et exécuter l'ensemble de ce projet. Notez la liste des commandes nécessaires pour cette compilation, vous en aurez besoin pour l'exercice 7.

Exercice 5. Vous trouverez dans le répertoire `Echange L2` le fichier `reines_gtk.zip`, qui contient un exemple de résolution du problème des n reines. Cette version utilise la bibliothèque LablGtk, que vous aurez besoin d'installer, et dont vous aurez besoin de consulter la documentation. Débrouillez-vous pour faire compiler et exécuter l'ensemble de ce projet. Notez la liste des commandes nécessaires pour cette compilation, vous en aurez besoin pour l'exercice 7.

Exercice 6. * Écrivez un fichier de commandes `make_clean.cmd`, qui servira à effacer tous les fichiers d'extension `.cmi` ou `.cmo`.

Exercice 7. * Convertissez les listes de commandes des exercices 1 et 3 en autant de fichiers `make.cmd`.

Exercice 8. Convertissez les listes de commandes des autres exercices en autant de fichiers `make.cmd`.